2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00069

faulTPM: Exposing AMD fTPMs' Deepest Secrets

Hans Niklas Jacob*, Christian Werling*, Robert Buhren, Jean-Pierre Seifert[†]
Technische Universität Berlin – SecT
[†]*also: Fraunhofer SIT*
{ *hnj, cwerling, robert.buhren, jpseifert* }*@sect.tu-berlin.de*

***Abstract*—Trusted Platform Modules (TPMs) constitute an integral building block of modern security features. Moreover, as Windows 11 made a TPM 2.0 mandatory, they are subject to an ever-increasing academic challenge. While discrete TPMs (dTPMs) – as found in higher-end systems – have been susceptible to attacks on their exposed communication interface, more common firmware TPMs (fTPMs) are immune to this attack vector as they do not communicate with the CPU via an exposed bus.**

In this paper, we analyze a new class of attacks against fTPMs: Attacking their Trusted Execution Environment (TEE) can lead to a full TPM state compromise. We experimentally verify this attack by compromising the AMD Secure Processor (AMD-SP), which constitutes the TEE for AMD's fTPMs. In contrast to previous dTPM sniffing attacks, this vulnerability exposes the complete internal TPM state of the fTPM. It allows us to extract any cryptographic material stored or sealed by the fTPM regardless of authentication mechanisms such as Platform Configuration Register (PCR) validation or passphrases with anti-hammering protection.

First, we demonstrate the impact of our findings by – to the best of our knowledge – enabling the first attack against Full Disk Encryption (FDE) solutions backed by an fTPM. Furthermore, we lay out how any application relying solely on the security properties of the TPM – like *Bitlocker's TPM-only* protector – can be defeated by an attacker with 2-3 hours of physical access to the target device. Lastly, we analyze the impact of our attack on FDE solutions protected by a *TPM and PIN* strategy. While a naive implementation also leaves the disk completely unprotected, we find that *BitLocker's* FDE implementation withholds some protection depending on the complexity of the used PIN. Our results show that when an fTPM's internal state is compromised, a *TPM and PIN* strategy for FDE is less secure than TPM-less protection with a reasonable passphrase.

1. Introduction

Trusted Platform Module (TPM) is a standard for a dedicated subsystem providing security primitives to a system. TPMs include a hardware random number generator, can measure and attest system state, securely generate cryptographic keys, and provide an interface for protecting sensitive data akin to a smartcard.

Initially, TPMs were exclusively implemented as discrete chips connected to the CPU via buses on the mainboard. However, CPU vendors introduced firmware TPMs (fTPMs) as an alternative implementation approach [24], [28], [42]. In contrast to discrete TPMs (dTPMs), fTPMs are implemented through code running in a Trusted Execution Environment (TEE) provided by the CPU vendor. While dTPMs are still found in higher-end devices, fTPMs' omnipresence in modern CPUs made them an affordable and convenient alternative for OEMs. In addition, the ever-increasing integration of TPM services into cryptographic APIs, e.g., *Platform Crypto Provider* in Microsoft Windows [31], fosters TPM usage across many layers and domains.

One of the more prominent use cases for a TPM is Full Disk Encryption (FDE), as it can secure *data-at-rest* without requiring the user to memorize a pre-boot passphrase. The standardized interplay of firmware and TPM, commonly called *Measured Boot*, verifies that the system was booted into a trusted state before FDE keys are released from the TPM to the Operating System (OS). Hereby, the system measures each firmware component and configuration value into the TPM, which keeps a cryptographically secure log of these measurements. Once the OS is in a trusted state, as reflected by the TPM's measurement logs, the TPM provides access to the disk encryption keys. Since the OS is now already in control of the system and its connected devices, it can effectively employ security measures like a login prompt to enforce access control to the user's data.

Threat model. We assume an attacker with prolonged physical access to a device, e.g., a stolen or lost laptop. While this puts the data on the laptop at immediate risk, TPM-based *Measured Boot* should be able to protect private data, such as a VPN key or FDE keys, from the attacker. This means only a properly booted runtime is able to access the sealed data. For this paper, vulnerabilities in the UEFI firmware or OS are out of scope.

Scenarios where this threat model becomes relevant are journalists protecting their sources or traveling employees carrying intellectual property on their company-provisioned laptop.

AMD's fTPM. While (d)TPMs were previously targeted at professional users and use cases only, the adoption of fTPMs highlights that TPMs of all sorts must undergo security testing. Moreso, since Windows 11 introduced a TPM 2.0 requirement [35], which in many devices like consumer-level laptops is available only as an fTPM.

In the past, dTPMs have been susceptible to attacks compromising their communication buses [7], [20], [50]. fTPMs, however, do not expose their communication with

[*] Both authors contributed equally.

DOI 10.1109/EuroSP57164.2023.00069

the CPUs, making this particular attack vector infeasible. Nevertheless, other attack vectors have come to light, especially concerning their execution environment.

While the dominant vendor for CPUs is Intel, AMD's market share has been at a record high recently [6]. The execution environment for fTPMs on AMD systems is the AMD Secure Processor (AMD-SP), which has recently been subject to physical and firmware attacks [14], [16]. In these works, the authors analyzed to what extent the Secure Encrypted Virtualization (SEV) technology is affected by these security issues. However, the authors neglected to analyze the impact of this Root of Trust (RoT) compromise on other critical security functionality provided by the AMD-SP.

In light of these attacks, we ask: What are the practical consequences of physical attacks against AMD's fTPMs?

1.1. Our Contributions

In this paper, we show that AMD's fTPMs are vulnerable to physical attacks against their execution environment: the AMD-SP. Our attack utilizes the AMD-SP's vulnerability to *voltage fault injection* attacks [14] to extract a chip-unique secret from the targeted CPU. This secret is subsequently used to derive the storage and integrity keys protecting the fTPM's non-volatile (NV) data stored on the Basic Input/Output System (BIOS) flash chip.

In contrast to the previous dTPM sniffing attacks, our approach exposes the complete internal TPM state of the fTPM. This allows us to extract any cryptographic material stored or sealed by the fTPM regardless of the authentication mechanisms, such as the measured system state or passphrases with anti-hammering protection. Additionally, once we have extracted the chip-unique secret through *voltage fault injection*, we only need to re-read the BIOS flash to defeat any TPM-based security measures.

To demonstrate the impact of our findings, we re-enable attacks against FDE solutions that rely solely on the TPM. Furthermore, we analyze how our findings impact an FDE key protected by a *TPM and PIN* strategy. We find that Microsoft *BitLocker's* FDE implementation is reduced to the security of a (TPM-less) *PIN-only* strategy. In contrast, a naive implementation leaves the disk completely unprotected once the TPM is defeated. The example of *systemd-cryptenroll*, a tool for enrolling hardware security tokens and devices into a LUKS2 encrypted volume, demonstrates this potential weakness.

In summary, our contributions are:

- We reverse-engineer the NV storage format of AMD's fTPM and the derivation of the chip-unique keys protecting its confidentiality and integrity.
- We leverage previously published hardware vulnerabilities on the AMD-SP to extract the cryptographic seeds used to derive the NV storage keys.
- Using the decrypted NV storage, we can extract any cryptographic secret and unseal arbitrary TPM objects protected with the fTPM.
- We use this ability to successfully attack Microsoft BitLocker's *TPM-only* key protector.
- We analyze the security of *TPM and PIN* protectors for FDE keys and describe how BitLocker withstands a compromised TPM when a strong PIN is used while a naive implementation does not.
- We publish all required tools to mount the attack in [27].

All security-relevant findings discussed in this paper were responsibly disclosed to AMD, Microsoft, and the systemd-cryptenroll maintainers. The systemd-cryptenroll maintainers quickly got back to us to discuss specific mitigation strategies.

2. Background

In this Section, we will introduce basic concepts necessary for our paper, i.e., Trusted Platform Modules, Full Disk Encryption in general, Microsoft BitLocker in particular, and the AMD Secure Processor.

2.1. Trusted Platform Modules

TPMs are secure cryptoprocessors specified by the Trusted Computing Group (TCG) featuring a hardware random number generator, secure generation of cryptographic keys, Platform Configuration Registers (PCRs) to measure a system's boot process, and secure storage/sealing of those keys or user-provided data. The 'TPM 2.0 Library specification' [48] (also known as ISO/IEC 11889) specifies how a TPM should interact with an external system. Two distinct types of TPMs can be distinguished based on their implementation: discrete TPMs (dTPMs) and firmware TPMs (fTPMs) [47]. The implementation type has different implications on the security level of the TPM. However, regardless of whether a dTPM, or a fTPM is used, the provided functionality is the same.
Both the security implications of the two implementation approaches, as well as the basic TPM functionality, are explained in the following sections:

2.1.1. Discrete TPM. dTPMs are dedicated hardware components implementing the TPM specification and are commonly built into commercial business laptops. dTPMs implement a degree of physical tamper resistance to protect the stored secrets from exposure and are therefore considered the most secure TPM variant [47]. Nonetheless, passive physical attacks have been demonstrated that target the communication channel between the dTPM and the rest of the system [50]. These can be used to, e.g., circumvent TPM-based FDE solutions that unlock a disk without the need for a passphrase or PIN.

2.1.2. Firmware TPM. fTPMs, on the other hand, implement TPM functionality mainly through software running in a Trusted Execution Environment (TEE) available to the CPU. Intel and AMD provide fTPMs with their Desktop CPUs, running on the Intel Management Engine (Intel-ME) and AMD-SP, respectively. Due to the integration of these coprocessors into the main die, they do not require external buses to communicate with the CPU. Therefore, sniffing attacks similar to the dTPM attacks would require costly on-die probing, which – to the best of our knowledge – has not been shown against fTPMs yet. On the other hand, fTPMs rely heavily on the security properties of the TEE.

2.1.3. Sealing.

A TPM can *seal* data, such as cryptographic keys or secrets, and regulate access to the data. For example, a user might generate an RSA key on the TPM, protect it with a passphrase, and limit it to be used as a signing key that can not be exported from the TPM, *binding* the key to the device. In such a scenario, the TPM acts like a smartcard.

Since TPMs have only a limited amount of non-volatile (NV) storage on the device itself, data is usually stored externally in a *sealed object* and can be loaded into the TPM when needed. While these external objects may reveal details about the kind of object that is sealed, the sensitive data, e.g., RSA private key, is encrypted and signed with storage keys only accessible to the TPM. The format of these external objects and the key derivation for their protecting keys are specified in the 'TPM 2.0 Library Specification' [48].[1]

The authorization options for sealed TPM objects are manifold and can be combined with 'and' and 'or' clauses to form complex authorization policies [9]. Two policies commonly used are PCR authorization and PIN/passphrase authorization with anti-hammering protections.

2.1.4. Platform Configuration Registers.

A PCR is a memory location in the TPM used to store a hash, e.g., a SHA256 digest. PCRs can only be updated by extending the existing value with a new value as follows [9]:

$$PCR[n] = hash\_alg(\ PCR[n] \mid\mid ExtensionValue\ )$$

The only way to reset a PCR is to reset the TPM, in which case it will reset to zero. This means that during a system's uptime, the PCR acts as a secure record of all extension values: Once a value has been recorded, one cannot 'un-record' it, i.e., force the PCR's value back to a previous state – unless the underlying cryptographic hash algorithm is flawed.

In the *measured boot* scenario – one of the most prominent use-cases for PCRs – each firmware component involved during boot is hashed and extended into an appropriate PCR *before* it is executed. In order to protect against malicious code runing early in the boot process, an operating system can record a set of known-secure PCR values and verify these PCRs on each boot. For example, on a PC platform, PCRs zero to seven record the system's boot process.

The TPM can further use the PCRs itself to authorize access to a TPM object. To check a PCR authorization policy, the TPM compares the current PCRs to a set of known values included with the policy. Such a policy can, e.g., protect a cryptographic key from being used when the system has been infected with a *Root- or Bootkit*.

2.1.5. PINs, Passphrases, and Anti-Hammering.

Another method of authorizing access to sealed TPM objects is a passphrase or Personal Identification Number (PIN) [9]. Of course, passphrase authorization is also commonly used for software-only protection, e.g., with SSH keys. However, TPM-based passphrase authorization can defend much more effectively against dictionary or brute-force attacks. The TPM can be configured to keep track of the failed passphrase authorization attempts and limit the amount or rate of authorization attempts. For example, Windows limits the authorization rate to one try every ten minutes once 32 failed attempts have been made [30]. This significantly limits the power of dictionary or brute-force attacks and even enables the use of lower entropy PINs to protect a TPM object.

2.1.6. Applications.

A significant application of TPMs is hardware-aided key management. The *tpm2-pkcs11* project [46], for example, exposes the TPM's functionality as a *PKCS #11* interface, which is a standardized API to access cryptographic services like smartcards [40]. With *tpm2-pkcs11* the TPM can be used to, e.g., hold and manage an SSH key protected with anti-hammering and bound to the system's TPM. Similarly, Microsoft uses the TPM as a hardware backend for their *Platform Crypto Provider* API in Windows [31]. Additionally, *Windows Hello* uses a device-unique asymmetric key-pair protected to authenticate a device with the identity provider (Windows). This key is sealed by and bound to the TPM of the device, as well as protected by PIN that authenticates the user towards the device [31]. Another prominent use case for TPMs is Full Disk Encryption (FDE), which we discuss at length in the next Section.

2.2. Full Disk Encryption

Full Disk Encryption (FDE) protects the confidentiality and integrity of *data-at-rest* of a computer, i.e., the contents of the computer's disks. This includes the computer's operating system, which means that the disk's encryption keys need to be available to the boot loader before it can load the operating system. One approach asks the user to enter a passphrase in a pre-boot environment and derive a key from this passphrase. However, to support changing the passphrase and allow multiple decryption methods, FDE tools like *BitLocker* or *LUKS* do not directly seal the key encrypting the *data-at-rest*, but store multiple encrypted copies of this key alongside the data [11], [36]. The respective *key-encryption keys* for these encrypted copies represent a method of decrypting the disk, e.g., a recovery key stored safely out-of-band, a passphrase, or a TPM-based decryption method. For a TPM-based decryption method, the FDE tools seal the *key-encryption key* with the TPM and protect it using the TPM's authentication mechanisms.

2.2.1. TPM-only strategy.

Here, the sealed *key-encryption key* is protected solely by a PCR policy (2.1.4). The PCR values necessary to unseal the key reflect a boot with a trusted firmware and boot loader configuration. If any part of the boot process is altered, e.g., by a UEFI Root- or BootKit, the PCR values reveal this change to the TPM, and the (possibly compromised) boot loader cannot unseal the *key-encryption key*. On the other hand, if the key can be unsealed – meaning the PCR values indicate a boot with valid and trusted firmware – the operating system's security measures enforce access policies to the disk's content and *data in use*, including the unsealed key.

[1] See '22 Protected Storage' of 'Part 1: Architecture' for the key derivation algorithms and, for the object format, `TPM2B_PUBLIC`/`TPM2B_PRIVATE` in 'Part 2: Structures'.

A notable distinction of this approach is its transparency toward to user. Although no interaction is required in the pre-boot environment, even an attacker with physical access cannot access the *data-at-rest* off the disks. However, a downside of *TPM-only* strategies for FDE is that they offer no additional security if the TPM protection can be broken, e.g., by a sniffing attack (3.2).

2.2.2. TPM and PIN strategy. A strategy that offers more protection than the *TPM-only* strategy against, e.g., sniffing attacks, is to include a PIN or passphrase. The TPM's anti-hammering features (2.1.5) compensate even the use of a lower-entropy PIN compared to a (TPM-less) *passphrase-only* strategy that can only protect against brute-forcing through a strong secret. If, on the other hand, an attacker gains access to the PIN or passphrase, the PCR policy still ensures that the key can only be accessed by the trusted boot loader running alongside the trusted firmware, leaving other (OS-level) protection mechanisms intact.

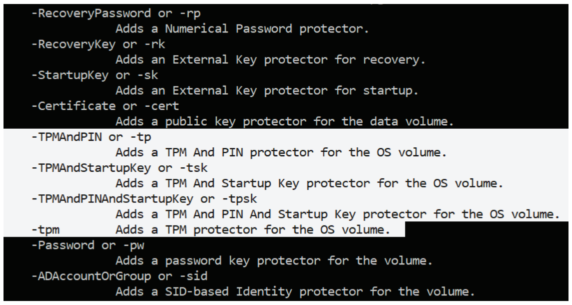
2.3. Microsoft BitLocker

BitLocker is a full-volume encryption feature by Microsoft integrated into Windows and available since Windows Vista. BitLocker aims to protect the confidentiality and integrity of *data-at-rest*, i.e., when the computer is powered off or in hibernate, from unauthorized access.

> BitLocker Drive Encryption is a data protection feature that integrates with the operating system and addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers. BitLocker provides the most protection when used with a Trusted Platform Module [...]. [29]

On Windows 11, BitLocker is enabled by default on systems with an enabled TPM and when a Microsoft Account is used during setup [51].

In the Windows runtime, BitLocker's deep integration into Windows makes the encryption and decryption of data completely transparent to user applications. During boot, BitLocker will provide an early (unencrypted) boot component handling the decryption of the remaining disk.



```
    -RecoveryPassword or -rp
                Adds a Numerical Password protector.
    -RecoveryKey or -rk
                Adds an External Key protector for recovery.
    -StartupKey or -sk
                Adds an External Key protector for startup.
    -Certificate or -cert
                Adds a public key protector for the data volume.
    -TPMAndPIN or -tp
                Adds a TPM And PIN protector for the OS volume.
    -TPMAndStartupKey or -tsk
                Adds a TPM And Startup Key protector for the OS volume.
    -TPMAndPINAndStartupKey or -tpsk
                Adds a TPM And PIN And Startup Key protector for the OS volume.
    -tpm        Adds a TPM protector for the OS volume.
    -Password or -pw
                Adds a password key protector for the volume.
    -ADAccountOrGroup or -sid
                Adds a SID-based Identity protector for the volume.
```

Figure 1. Available protector types for Microsoft BitLocker on Windows 11, as displayed by 'manage-bde'. TPM-related protectors are highlighted.

2.3.1. Key management. All data on disk is encrypted using the Full Volume Encryption Key (FVEK). The FVEK is stored encrypted by the Volume Master Key (VMK) in an unencrypted portion of the volume. The VMK is stored encrypted by several so-called *(key) protectors* that can be seen in Figure 1, one of which is, by default, the numeric recovery key a user is requested to print out during setup. The encrypted copies of the VMK – by default, a TPM-protected VMK and a recovery key-protected VMK – are stored in an unencrypted portion of the BitLocker-protected volume. They can be managed on Windows through a Microsoft-provided PowerShell tool ('manage-bde') and Linux with the third-party tool *Dislocker* [8]. *Dislocker* is furthermore able to mount a BitLocker-encrypted volume, given an unencrypted VMK.

2.3.2. TPM-based protectors. Besides protector modes like a passphrase or USB key, BitLocker provides multiple protectors facilitating a TPM. In fact, BitLocker, by default, uses the TPM's boot integrity measurements exclusively to ensure that an only untampered BitLocker runtime can access the TPM-protected VMK.

During setup, BitLocker takes the VMK and *seals* it with the current TPM state represented by the PCR register values 0, 2, 4, and 11. It then stores a handle to this *sealed* TPM object in the volume's unencrypted BitLocker header (as explained in more detail in 5.2). After setup and during each boot, BitLocker relies on the TPM to *unseal* (2.1.3) the VMK. However, the TPM will only do so if the PCR register values match those saved in secure storage and defined during the BitLocker setup.

This makes the *TPM(-only)* protector entirely transparent for the user during boot, as it requires no additional user interaction when the PCRs are in the expected state. If not, BitLocker will fall back on the recovery protector. In order to decrypt it, it will prompt the user to enter the recovery key.

2.4. AMD Secure Processor

The AMD Secure Processor (AMD-SP) is a dedicated security co-processor part of all recent AMD Systems-on-a-chip (SoCs), including the Ryzen CPUs. Since its introduction in 2013, the AMD-SP (formerly known as Platform Security Processor (PSP)) has acted as the Root of Trust (RoT) of the SoC [28]. Its responsibilities on the Ryzen platform include the early SoC initialization, starting and initializing the secure boot chain, providing a TEE, and hosting the fTPM application [4], [12], [19].

2.4.1. Early Boot. The AMD-SP boots before the main X86 cores of the AMD SoC [13]. As illustrated in Figure 2, the first boot stage to run on the AMD-SP is an immutable Read-Only Memory (ROM) boot loader. After some minimal system initialization, the ROM boot loader loads the AMD Root Key (ARK) from the BIOS flash chip and verifies the key by comparing its SHA256 digest to a known value that is part of the AMD-SP's immutable ROM [14]. Once verified, the AMD-SP loads its next boot stage, the so-called *off-chip* boot loader, from the BIOS flash chip and verifies it using the ARK. Finally, the *off-chip* boot loader initializes various components of the SoC and executes other system initialization routines like DRAM training.

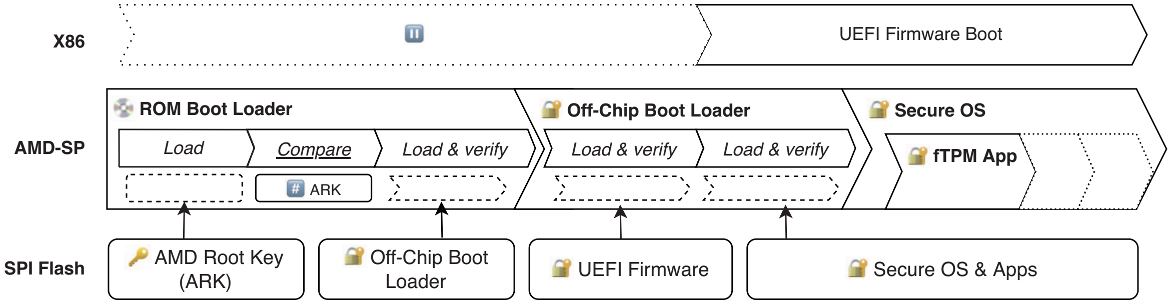


Figure 2. Early boot of a modern AMD CPU. A key denotes a public key, a pound sign a hash, and a lock a signed element.

After the system initialization, the AMD-SP loads the initial boot firmware into X86 memory and starts the X86 cores [12]. Once the X86 cores are running, the *off-chip* boot loader is replaced with a microkernel. This microkernel acts as a TEE for, among other applications, the fTPM implementation.

2.4.2. Firmware. Besides the ROM boot loader, all firmware executed on the AMD-SP is loaded from BIOS flash. Although the general format is standardized by the Unified Extensible Firmware Interface (UEFI) volume standard, the AMD-SP's code is stored in a proprietary file system. The open-source tool *PSPTool* [49] aims to parse and modify these *firmware file systems*. It has been used by Buhren et al.'s security-related publications about AMD processors [12], [14], [16].

2.4.3. Cryptographic Co-Processor (CCP). The AMD-SP features a dedicated hardware component, the so-called Crypto Co-Processor (CCP), that allows offloading various cryptographic operations and is available to both the AMD-SP and the X86 CPU. Although not officially documented by AMD, the CCP Linux driver gives an insight into its functionality [1]. The CCP features a local memory space commonly called the Local Storage Buffer (LSB), which can hold keys or other data for cryptographic operations.

2.4.4. Attacks. In [12], Buhren et Eichner present their emulation efforts of the PSP and disclose a critical vulnerability in the ROM boot loader of supposedly all Zen 1 and Zen Plus CPUs. A stack-based buffer overflow caused by user-provided data from the Serial Peripheral Interface (SPI) flash yields privileged code execution in the *ROM Boot Loader* (as seen in Figure 2). Unfortunately, since the vulnerability lies in the on-chip ROM of the PSP, AMD cannot issue any fix for this vulnerability.

However, in this paper, we use a voltage fault injection attack to gain code execution on the AMD-SP of the newer Zen 2 and Zen 3 CPU generations as introduced by Buhren et al. in [14]. This attack leverages the Serial Voltage Identification Interface 2.0 (SVI2) bus, allowing the AMD SoC to update its supply voltages dynamically. By injecting packets onto this bus, an attacker causes a short drop in the AMD-SP's supply voltage and induces a fault in the AMD-SP. With a carefully timed fault injection, Buhren et al cause the *Compare* operation illustrated in Figure 2 to accept a modified ARK previously placed on the BIOS flash chip. With *PSPTool's* capabilities to replace and resign various AMD-SP firmware components, this fault injection attack can be used to gain code execution in various stages of the AMD-SP's runtime.

3. Related Work

3.1. TPM Attacks

3.1.1. Side-channel attacks against fTPMs and dTPMs. The most recent academic attack on TPMs is [37]. Moghimi et al. perform a black-box timing analysis of TPM 2.0 devices and find secret-dependent execution times during signature generation. These timing leakages are discovered on both an Intel fTPM and a dTPM by STMicroelectronics.
Both Intel and STMicroelectronics have released firmware updates addressing the vulnerabilities [52].

3.1.2. Power management attacks against the TPM 2.0 specification and tboot. In [22], two sorts of TPM attacks regarding power management are reported where Han et al. find a way to reset and forge the TPM's PCRs values. One vulnerability targets a grey area design flaw in the TPM 2.0 specification. The other exploits an implementation flaw in the most popular measured launch environment used with Intel's Trusted Execution Technology (TXT), 'tboot'.
While the authors provided a patch to mitigate the latter, they contacted and reported their findings of the former to Intel, Dell, Gigabyte, and Asus.

3.1.3. AMD fTPM trustlet code execution attack. In an earlier public disclosure of AMD's fTPM security [19], Cohen finds a stack-based buffer overflow in the fTPM trustlet running on the PSP. The vulnerability is exploitable through user-controlled data in a TPM 2.0 call and allows full control of the program counter. By applying additional exploit techniques like return-oriented programming, this vulnerabilty can make it possible to break the fTPM's security guarantees.
AMD issued firmware updates to mitigate the vulnerability [2].

3.1.4. LPC sniffing attacks against dTPMs. Even though dTPMs are tamper-resistant devices, the LPC bus connecting it to the main CPU is not. In [50], Winter et Dietrich show that passive attacks against dTPMs's bus communication can be mounted with reasonably inexpensive equipment. Moreover, active attacks allow the authors to circumvent any security mechanism provided by the TPM, e.g., the chain of trust.

3.2. BitLocker Attacks

Two of these attacks remain problematic for BitLocker until today:

1) Microsoft recommends defeating any *power management-based attacks* (3.1.2) by "disabl[ing] Standby power management, and shut[ing] down or hibernat[ing] the device before it leaves the control of an authorized user" [30].

2) To protect against any *LPC sniffing attacks against dTPMs* (3.1.4), Microsoft advises using a TPM with PIN protector instead of a purely PCR-based TPM protector [30].

3.2.1. Cold boot attack against RAM. *Cold boot attacks* are a physical side-channel attack in which an attacker performs a memory dump of the RAM by performing a hard reset of the target machine. They rely on the data remanence property of DRAM and SRAM that allows retrieving memory contents seconds to minutes after power off. Since BitLocker stores the essential key material in memory, this attack can be mounted regardless of the used BitLocker protectors.
Full memory encryption, as implemented by AMD [3] and proposed by Intel [25], can potentially mitigate cold boot attacks.

3.2.2. Drive-by DMA attacks. In the past, BitLocker was a popular target for Direct Memory Access (DMA) attacks using FireWire [10]. As it allows an attacker to retrieve BitLocker keys directly from memory, its nature is similar to 3.2.1 but with less complex hardware requirements. Such attacks are still relevant [43] but actively mitigated by leveraging the system IOMMU to implement kernel DMA protection [32].

3.2.3. Dictionary attacks. Decrypting the BitLocker VMK was optimized for GPUs in [5]. However, cracking it in such a fashion remains a costly and time-consuming task.

4. fTPM Attack

In this section, we present our attack on AMD's proprietary fTPM, which allows us to decrypt sealed TPM objects regardless of their authorization policy (2.1.3).

To carry out our attack, we execute a small payload that leaks a *chip-unique secret* from the CPU (4.3). This secret is used to derive the encryption and signature keys for the fTPM's *non-volatile data*, which is stored on the BIOS flash chip (4.2). As a result, we now have the ability to decrypt or modify the fTPM's *non-volatile* state, which we use to get access to the *storage keys* of sealed TPM objects (4.4).

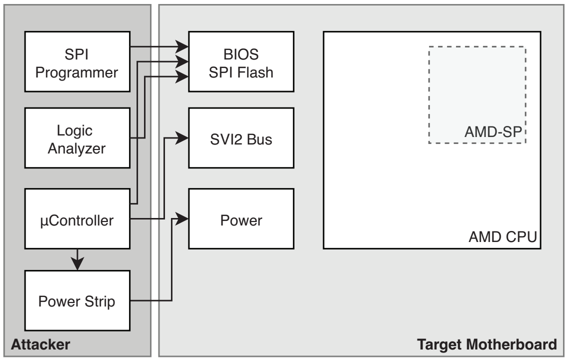


Figure 3. Physical connections necessary for the attack

Hardware Access. In order to analyze and attack AMD's fTPMs, we use the *voltage fault injection attack* presented by Buhren et al. [14]. This attack (2.4.4) allows us to gain code execution during various stages of the AMD-SP's firmware, including the fTPM application (2.4.1). The attack requires access to the motherboard of the target system (4.1), particularly its SPI bus and voltage regulators. After leaking a CPU's *chip-unique secret*, no more glitching is required. To attack this CPU, we now merely need access to the BIOS flash, which can simply be read by accessing the motherboard's SPI bus.

Older CPUs. Our attack targets AMD Ryzen CPUs of the microarchitecture generations Zen 2 & 3, which use a common fTPM implementation. The older Zen (1) and Zen + CPUs use a different fTPM implementation and, particularly a different non-volatile storage format. If an attacker reverse engineers this storage format and its key-derivation algorithm, the same attack approach does apply. Since there are code-execution attacks for these CPUs that do not need fault-injection whatsoever (2.4.4), they should be considered even more vulnerable.

4.1. Voltage Fault Injection Attack

Recall the AMD-SP's boot-process (2.4): An immutable ROM bootloader loads and verifies the AMD Root Key (ARK), which is used to verify all further firmware components, including the *off-chip bootloader* that is executed after the ROM bootloader. Buhren et al.'s *voltage fault injection attack* causes the AMD-SP to accept an invalid ARK and thus enables an attacker to replace and resign various firmware components on AMD Epyc CPUs. In this section, we introduce details about this attack and highlight changes that were necessary to apply the attack to Ryzen CPUs and laptops.

4.1.1. Hardware Setup. To leverage the AMD-SP's susceptibility towards *voltage fault injection*, an assortment of readily available hardware is required:

SPI Programmer In order to read and write the BIOS flash chip, an SPI flash programmer is needed.

Attack µController A small microcontroller board injects the *voltage fault* and generates the fault injection trigger. As described by Buhren et al. [14], a driver IC

Description	Cost in USD
Teensy 4.0 Development Board	∼ 20$
SPI flash programmer	∼ 15$
Logic Analyzer	∼ 15$
Driver IC and additional resistors	∼ 5$
Flash chip test clip	∼ 5$
Test probes with spring-loaded pins	∼ 120$
Controllable Power Relay	∼ 15$
Total	∼ 195$

TABLE 1. TOTAL HARDWARE COSTS FOR THE ATTACK AT TIME OF WRITING

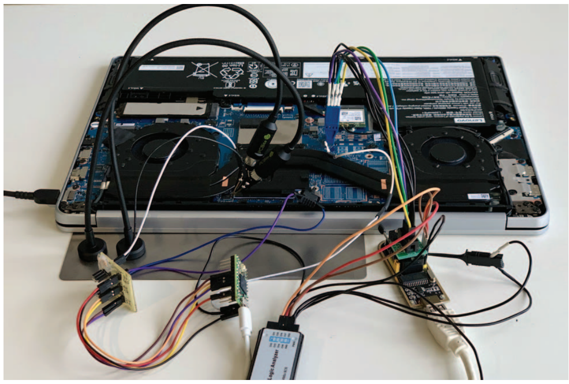


Figure 4. Attack setup on a Lenovo Ideapad 5 Pro-16ACH6

and some additional resistors aid the microcontroller with its SVI2 bus injection.

Reset Method Since the attack also requires frequent restarts of the device under attack, the microcontroller needs to be able to reset the target. In the case of a desktop motherboard, the original glitch attack's method using the ATX case header's reset pin can be used [14]. For laptops, which generally do not expose an ATX case header, a relay can be used to control the laptop's power supply while disconnecting any batteries.

Logic Analyzer Buhren et al. use the SPI bus to extract data from the AMD-SP [14]. To capture this data, a logic analyzer is also connected to the SPI bus and records the communication.

Figure 3 illustrates all necessary connections to the target motherboard.

For the additional connections to the motherboard's SVI2 bus, we used test probes with spring-loaded tips. With these, we could reliably connect our injection hardware without soldering, even to adjacent pins of the TQFN packages used by all target Voltage Regulators (VRs) that we attacked.

The final attack setup can be seen in Figure 4. At the time of writing, the total hardware cost amounts to under 200 USD, more than half of which is spent on the test probes (Table 1).

4.1.2. Gaining Code Execution. Once the fault injection hardware is connected, the attack by Buhren et al. consists of a manual *parameter determination* phase and a brute-force search for a final *delay* parameter [14]. The first step requires around 30 minutes of manual attention at the moment, we believe it can be automated, as algorithm-like descriptions of the *parameter determination* process can already be found in the attack's supplementary code repository [14], [15]. The attack's second phase consists of a loop of repeated attack attempts to search for the last to-be-determined parameter and execute the attack's payload.

4.1.3. Payload creation. Our payloads consist of a short *ARMv7a* assembly section (the AMD-SP's architecture [28]) to bootstrap our C-code payload. The necessary hardware details, e.g., the MMIO interfaces of the AMD-SP's SPI controller or the CCP (2.4.3), can be found in Buhren et al.'s supplementary repository [15]. In order to finally gain code execution, we create our own RSA key pair and replace the ARK on the BIOS flash chip with our key. We then replace the off-chip boot loader on the BIOS flash chip and resign it with our ARK replacement key. Once we successfully inject a fault and the AMD-SP accepts our ARK, it will load and execute the modified off-chip boot loader.

To make cryptographically consistent changes to the AMD-SP's file system, we rewrote parts of the open-source tool *PSPTool* [49] that check and re-create signatures when a key or other firmware file is replaced. Our modifications to *PSPTool* have since been upstreamed to the current version of the tool [49]. In addition to executing a payload instead of the *off-chip boot loader*, we can also patch existing firmware components, resign the image, and boot the whole AMD SoC with the modified firmware.

4.2. Non-Volatile Storage

The non-volatile (NV) state of AMD's fTPM is stored on the motherboard's BIOS flash chip. To protect against an attacker with read or write capabilities to the BIOS flash chip, the confidentiality and integrity of the NV state are cryptographically protected. We reverse-engineered the data structures of these files, as well as the storage and integrity key derivation algorithms for Ryzen CPUs of the Zen 2 and Zen 3 microarchitecture generations[2].

The fTPM's non-volatile state can be found in a file stored alongside the AMD-SP's firmware on the BIOS flash chip. *PSPTool* labels this file `NV_DATA`. Although the `NV_DATA` file's sensitive data is encrypted, its metadata and structure can be understood without access to the encryption keys. `NV_DATA` files are divided into two 64 KiB sections featuring append-only data structures (left side of Figure 5). Once both sections are filled, the older of the two sections is overwritten and it is ensured that the new file contains all the data necessary for the fTPM. Each section consists of a header and multiple entries, each of which is associated with a *context* that indicates the entry's usage. Furthermore, the entries of each context are ordered by an increasing *sequence* number.

Each entry consists of a variably sized body encrypted using AES128 in counter mode. The entry's body can be subdivided into up to seven variably sized fields. The integrity of each entry is protected by an HMAC-SHA256 MAC over the encrypted body, the IV for the encryption cipher, and the unencrypted field-length specification (see

[2] This storage format differs for Zen 1/+-based systems.

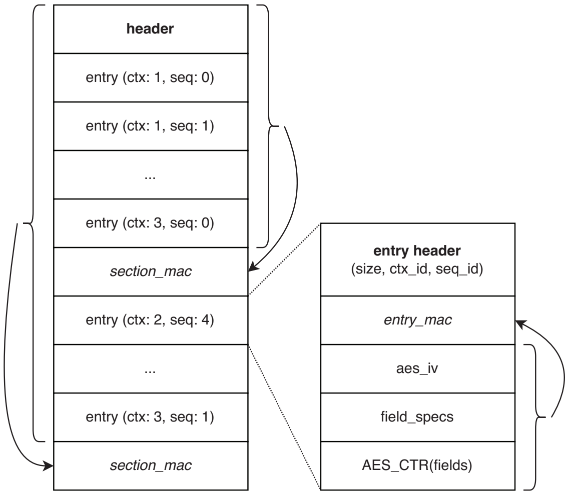


Figure 5. Format of an NV section (left) and entry (right)

right side of Figure 5). In addition to this *Encrypt-then-MAC* protection of each entry, the section data is protected by occasional HMAC-SHA256 message authentication codes (MACs) over the whole section contents up to the latest entry (left side of Figure 5). A tool for parsing (and decrypting) this NV storage format is presented in Section 4.3.2.

4.2.1. NV storage key derivations. To reverse-engineer the key derivation process for both the *storage (AES) key* and the *integrity (HMAC) key*, we used the ability to patch and replace arbitrary firmware components of the AMD-SP (4.1). The fTPM application runs as an application in the *SecureOS* microkernel (2.4.1). Another application, labeled `DRIVER_ENTRIES` by the *PSP-Tool*, implements drivers for device-specific functionality, like the CCP or SPI bus. We statically analyzed the `DRIVER_ENTRIES` binary and created a modified version that logs every cryptographic operation, including its inputs and outputs (see Figure 6), to the AMD-SP's SPI bus.

The storage and integrity keys are derived from a $128\,\mathrm{bit}$ secret unique to each CPU. This chip-unique secret is held by the CCP of the AMD-SP. It is present at address zero of the LSB (2.4.3). Across all devices we tested, the derivation process (illustrated in Figure 6) was consistent:

1) The secret is used as key in an *AES128* decrypt operation with a constant as ciphertext.
2) From the AES operation's output, two $256\,\mathrm{bit}$ values are derived using a NIST specified key derivation function (KDF)[3], where "AES key for wrapping data" and "HMAC key for wrapping data" are used as the *label* inputs.
3) Into each of these values, the signing key of the fTPM application is mixed. This is done by computing the SHA256 digest of the RSA key's modulus and calculating the HMAC-SHA256 of this digest with the secret as a key.

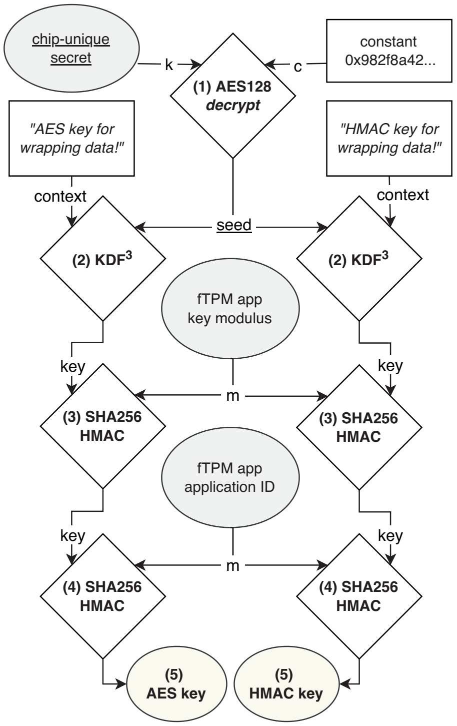


Figure 6. Key derivation used for the NV data keys

4) Similarly, the *id* under which the fTPM application is run in the *SecureOS* is mixed in. The resulting values are the HMAC-SHA256 of the $128\,\mathrm{bit}$ id, used as message, and the respective secrets, as keys.
5) Finally, the first value is truncated to its first $128\,\mathrm{bits}$ and used as the *AES storage key*, while the full $256\,\mathrm{bits}$ of the second value are used as the *HMAC integrity key*.

4.3. Secret Extraction

We extracted the output of each step of the key derivation process by booting the system with a patched `DRIVER_ENTRIES` binary and analyzing the traced cryptographic operations. However, this entails the challenge of leaving the boot functionality intact.

In order to remove the need to fully boot the system and to make our attack work with other versions of the `DRIVER_ENTIRES` binary, we built a payload (4.1.3) that directly computes the output of step (1) of the key derivation process and writes the result to the SPI bus. Then, using the fault injection attack, we can execute the payload in place of the off-chip boot loader (4.1) and, with the logic analyzer, extract the seed value (underlined in Figure 6) from the SPI bus.

This extracted seed is all that is necessary to decrypt a Zen 2 or Zen 3-based fTPM's internal state from a BIOS image. However, we additionally provide the means to leak the unmodified *chip-unique secret* from pre-Zen 3 CPUs:

4.3.1. Extracting the chip-unique secret. The chip-unique secret (underlined in Figure 6) used in the fTPM's key derivation is contained in a read-protected area of the

[3] KDF in counter mode with HMAC-SHA256 as *psoudorandom function* and an empty context, specified in NIST's SP 800-108 [18].

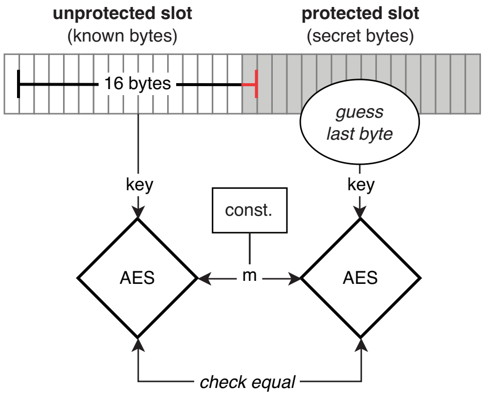


Figure 7. Our LSB extraction technique illustrated

CCP (2.4.3). This is presumably meant to add a layer of protection by preventing the secret from being extracted from the LSB. But, due to an oversight in the CCP's interface design, we are able to extract the entire contents of the LSB for Zen 1, Zen +, and Zen 2 CPUs, including read-protected areas. This extraction is possible due to the allowed use of unaligned key addresses in AES crypto operations.

Usually, the LSB is accessed in multiples of 16 byte regions, so-called *slots*, but the CCP allows AES operations with arbitrarily aligned keys. In particular, the CCP allows us to execute an AES128 operation with a key that is contained partially in a read-protected and partially in an unprotected slot. We use this to execute an AES128 encrypt operation with a key of which 120 bits (15 bytes) are known, and only 8 bits (1 byte) are unknown. Furthermore, we can choose the operation's input and have access to its output. This enables us to brute-force the unknown byte by comparing the AES operation's output to the 256 possible outputs that an AES operation with the given input and possible key values can have (see Figure 7). By shifting the key-window one byte at a time and repeating the single-byte brute-force attack, we can extract the entire read-protected slot.

On Zen 3 CPUs, AES operations with an unaligned key result in an error, foiling this extraction method for AMD's latest generation Ryzen CPUs. Further inquiries into the CCP interface did not yield similar vulnerabilities for Zen 3. Nevertheless, since the ciphertext for operation (1) of the key-derivation algorithm (Figure 6) is constant, leaking the output of this operation, as described in Section 4.3, is sufficient to achieve the same goal on Zen 3 CPUs.

4.3.2. Tooling. We included the payloads that implement both methods of extracting key derivation secrets in the paper's supplementary code repository [27]. Additionally, to parse and decrypt the fTPM's NV storage, we developed the Python tool *amd-nv-tool*, whose code we also published alongside the paper. *amd-nv-tool* parses the unencrypted structure and metadata of the `NV_DATA` file, derives the storage and integrity keys, and finally outputs the NV storage's contents in a JSON representation.

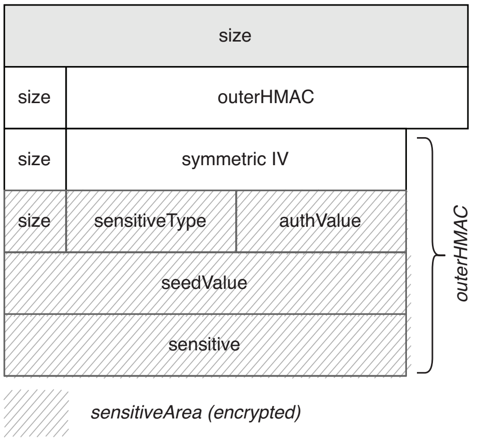


Figure 8. Format of `TPM2B_PRIVATE` object

4.4. TPM Object Decryption

At this point, we have an attack primitive to extract a chip-unique secret for a CPU and decrypt the fTPM's non-volatile storage with this secret. One application of this decrypted TPM state is to decrypt sealed TPM objects (2.1.3). Externally stored (sealed) TPM objects consist of a public and a private part. The public part communicates metadata about the object, e.g., the object authorization policy, and uniquely identifies the object (`TPM2B_PUBLIC` in "Part 2: Structures" of the TPM specification [48]). No part of this public part is encrypted or its integrity protected.

In contrast, the object's private portion is protected with an *encrypt-then-MAC* approach involving a *symmetric* and an *HMAC* key (see Figure 8). The encrypted part of the object consists of structural metadata, an authentication value, a seed, and the object's sensitive data. The authentication value is used for PIN or passphrase authentication, while the seed adds entropy to the object and can be used to seal child objects of this object. In the sensitive data part, we find the sealed data or, in the case that our object is a key, its the symmetric or private key.

4.4.1. Deriving the sealing keys. As described in Section "23 Protected Storage Hierarchy" of "Part 1: Architecture" of the TPM specification [48], TPM objects are organized in a tree of objects, where each object is 'sealed' by its parent. This is realized by deriving the sealing keys of an object from its public portion together with the parent's seed value (see "22 Protected Storage" of "Part 1: Architecture" of the TPM specification [48]). The root, or *primary*, objects of these trees can be deterministically generated from a persistent seed value but are, per default, cached in the TPM's non-volatile storage [9].

We used this caching to our advantage and implemented an unsealing tool that searches all consecutive 256 bit values in the decrypted non-volatile fTPM state for the correct *primary* seed. To check whether a possible seed value is the desired primary seed, our tool derives the corresponding HMAC key and tries to verify the HMAC of the object's private portion. Once this verification succeeds, we use this *primary* seed to derive the symmetric key and decrypt the object's private part.

System	CPU
Lenovo Ideapad 5 Pro 16ACH6	Ryzen 5600H (Zen 3)
Asrock 520M-HDV	Ryzen 5600X (Zen 3)
Asrock 520M-HDV	Ryzen 3600 (Zen 2)

TABLE 2. Targets verified to be vulnerable to our fTPM attack

This means, with access to the decrypted non-volatile fTPM state, we are able to decrypt any TPM object whose primary key is cached on the fTPM, *regardless* of the object's authentication policies. We implemented the key derivation and decryption detailed above in our `amd_ftpm2_unseal` tool, published alongside this paper.

4.5. Results

To summarize, we consider an attacker in possession of a victim device, e.g., a laptop, wants to carry out the attack described in this paper to attack any fTPM-based security measure. The necessary steps they will have to mount are as follows:

1) Backup the BIOS flash image using an SPI flash programmer
2) Connect the fault injection hardware and determine the attack parameters (4.1)
3) Compile & deploy the payload extracting the key derivation secret (4.3)
4) Start the logic analyzer to capture the extracted key derivation secrets via SPI
5) Start the attack cycle on the target machine until the payload was executed successfully
6) Parse & decrypt the NVRAM using the BIOS ROM backup and payload output with *amd-nv-tool*
7) Extract and decrypt TPM objects protected by this fTPM with *amd_ftpm_unseal*

Note that once the seed or chip-unique secret has been extracted (steps 2 to 5), the attack can be re-mounted quickly by reading the BIOS flash image, parsing the contained NVRAM, and decrypting external TPM objects (steps 1, 6, and 7). This process takes us about 15 minutes on a standard laptop.

We successfully executed the attack end-to-end on the hardware listed in Table 2. After having gained experience with the attack, we are able to perform the full attack on a new device within two to three hours. All necessary tools, as well as sample intermediate data, are available in the supplementary repository [27].

Furthermore, we were able mount the LSB secret extraction (4.3) on Zen 1 and Zen + CPUs by using the less complex ROM boot loader attack (2.4.4). A simplified attack (without the need for voltage glitching) can, therefore, compromise Zen 1/+-based fTPMs, if an attacker conducts the necessary reverse engineering efforts.

5. Case Study: BitLocker

To evaluate the severity of our fTPM attack on TPM-based applications, we conducted a case study on the implementation of *BitLocker's TPM-only* and *TPM and PIN* protectors (2.3). Since a *TPM-only* strategy does not apply any other means of protection than the TPM, we expect all such protection mechanisms to be broken by our attack.

However, as shown in Figure 1, BitLocker provides additional authentication factors which can be combined with a TPM: *PIN*, i.e., a 4- to 20-digit numeric or alphanumeric passphrase, and *StartupKey*, storing part of the encryption key on a USB flash drive. Since the *StartupKey* protector, according to Microsoft documentation, does not make any use of the TPM [33] but supposedly acts as another layer of cryptographic protection, it does not apply to our fTPM attacks and is not subject to our case study.

5.1. Threat Model

BitLocker "addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers" [29]. Since all these events regard the *physical* state of the protectee's device, BitLocker's goal is to secure data from an attacker with *physical* access to the device. However, Microsoft differentiates two types of (physical) attackers:

1) An **opportunistic attacker** "does not use destructive methods or sophisticated forensics hardware/software" [...] "Physical access may be limited by a form factor that does not expose buses and memory."
2) A **targeted attacker** has "plenty of time; this attacker will open the case, will solder, and will use sophisticated hardware or software."

Our threat model considers a targeted attacker who has gained prolonged access to the device, e.g., a thief targeting a company laptop. This attack type is commonly referred to as an *Evil Maid attack*. However, as we will discuss in detail in Section 6.3.2, for Zen 1 and Zen + systems, our attack may be suitable for an opportunistic attacker as well.

5.2. TPM-only Protector

To apply our attack described in Section 4, we first need to get ahold of the TPM objects that facilitate *BitLocker's* TPM protection. The *Dislocker* toolchain [8] can parse the unencrypted headers and metadata of a *BitLocker* volume and decrypt a volume given one of the volumes VMKs. For a *BitLocker* volume protected by a *TPM-only* protector, the metadata corresponding to that protector consists of a single *datum* labeled "`TPM-encoded`". This *datum* contains the sealed TPM object containing a VMK and the metadata required to fulfill the PCR policy authentication of the object.

In detail, the datum contains the concatenation of the TPM object's private and public portion (4.4), to which another data structure specifying the PCR policy is appended. As described in Section 4.4.1, with the decrypted non-volatile fTPM state, our `amd_ftpm2_unseal` tool can extract the wrapping keys for the sealed object and decrypt its contents. The unsealed TPM object contains another *BitLocker datum*, the last $256\,\mathrm{bits}$ of which are the disk's VMK. To verify our exploit, we mounted and decrypted the *BitLocker* volume using the `dislocker` toolchain's `dislocker-fuse` command [8].

5.3. TPM and PIN Protector

The *TPM and PIN* protector consists of multiple *datums* in a volume's BitLocker metadata. These include *datums* used for key stretching – similar to those used for a recovery password – in addition to a TPM encoded *datum* like the one of the *TPM-only* protector. In contrast to the *TPM-only* protector, the unsealed *datum* does not directly contain the VMK, but an AES-CCM encrypted *datum*.

Once the TPM protections have been circumvented, it is still necessary to brute-force, or otherwise acquire, the PIN. We assume that BitLocker uses this scheme to provide reasonable protection even in case the TPM's security properties can be subverted:

> For some systems, bypassing TPM-only may require opening the case, and may require soldering, but it could be done at a reasonable cost. Bypassing a TPM with a PIN protector would cost much more, and require brute forcing the PIN. With a sophisticated enhanced PIN, it could be nearly impossible. [30]

Although recommended by Microsoft for some scenarios, we will discuss in 6.4.2 that it is not straightforward to step up from the default *TPM-only* to a *TPM and PIN* protector.

5.3.1. Brute-forcing the PIN. The PIN-based key derivation involves a 128 bit salt and 1 048 576 rounds of SHA256, which limit the speed of a brute-force attack to around 1 000 PINs/passwords per second on a GPU [5]. In contrast, on a dTPM, the hardware-aided anti-hammering mechanism limits the authorization rate to one attempt every ten minutes [34].

		time to brute-force	
PIN/password	min-entropy	fTPM	dTPM
4 digits	2^9	0.5 sec	3.5 days
10 digits	2^{15}	33 sec	7.3 mo
10 characters	2^{21}	34 min	41 yr
20 characters	2^{36}	2.1 yr	$1.3 \cdot 10^6$ yr

TABLE 3. ESTIMATED BRUTE-FORCE TIMES BASED ON NIST'S PASSWORD GUESSING ENTROPY [17]

As can be seen in Table 3, the additional security provided by a numeric PIN – in case of a compromised fTPM – is negligible, while a 4-digit PIN already defeats a traditional evil maid attack if a dTPM is used. On the other hand, a properly chosen passphrase can provide an adequate level of security even with a compromised fTPM.

5.3.2. Alternative implementations. The naive approach to a *TPM and PIN* protector is to only rely on the TPM's authentication mechanism to verify the PIN. As detailed in Section 4.4, these authentication mechanisms do not mean the PIN/passphrase are involved in the sealed TPM-object's encryption and therefore do not impose any restrictions on our attack.

An FDE tool that does not implement a *TPM and PIN* strategy with a *defense-in-depth* approach is the open-source tool *systemd-cryptenroll*. The *systemd-cryptenroll* tool is part of the widely adopted *systemd* project and acts as a management tool for encrypted disks conforming to the popular *LUKS* standard [44], [45]. Support for TPM based protections has only been introduced recently and includes a *TPM-only* and a *TPM and PIN* strategy [21], [41]. Our analysis of the *systemd-cryptenroll* code shows that a randomly generated 256 bit secret is directly sealed by the TPM, protected either by a PCR policy only or additionally a PIN. The so-called *LUKS keyslot* (analogous to *BitLocker's* VMK) is then encrypted with the base64-encoded secret as passphrase.

Once the NV state is decrypted, the LUKS key is directly accessible and no brute-forcing is necessary.

To mitigate this issue, we recommend including the PIN in the passphrase protecting the *LUKS keyslot*. With this approach, we can protect the disk and PIN with the brute-force resistant key-derivation mechanism of the *LUKS keyslot*, even if the TPM encoded secret was leaked by an attack like the one described in this paper. We proposed this approach to the maintainers of *systemd-cryptenroll*.

5.4. Results

Our case study shows that *TPM-only* protection mechanisms for FDE are ineffective when the TPM's internal state can be extracted. In particular, we demonstrated this with *BitLocker's TPM-only* protector using our attack against AMD's fTPM. Furthermore, *BitLocker's TPM and PIN* protector retains the protection that a *PIN-only* strategy would offer, which is basically negligible in the case of a numeric PIN.

With a passphrase, however, the same level of security that the *passphrase-only* protector offers is retained, thanks to the additional layer of encryption applied to the VMK before it is sealed by the TPM. This reveals a need for careful consideration when implementing a *TPM and PIN* strategy. On a vulnerable fTPM, such a strategy without additional brute-force protection may be less secure against a targeted attacker than a *PIN/Passphrase-only* strategy with a similarly strong PIN or passphrase.

6. Discussion

In this section, we evaluate the feasibility and impact of our attack, propose potential mitigations, and, in light of publicly known TPM attacks, discuss important considerations for the use of BitLocker with a TPM, as well as the secure implementation of systems and applications relying on a TPM in general.

Our attack model assumes that an attacker has prolonged physical access to the target system. Even though these are strong prerequisites, they are well within the threat model of typical applications using the TPM, e.g., Microsoft BitLocker, detailed in Section 5.1. For example, a typical scenario is when an attacker has stolen a laptop with valuable company secrets that is protected with Full Disk Encryption (FDE).

6.1. Requirements

In order to mount the key extraction, the attacker needs to be able to execute a custom payload on the AMD-SP. Although we demonstrated the attack on newer Zen 2/3 systems, code execution on Zen (1)/+ systems can be achieved with a simpler setup.

6.1.1. Zen 2/3 (Glitch attack). As detailed in Section 4.1.1, the hardware required to carry out the attack is readily available and amounts to under 200 USD.

While the attack requires manual examination of each target motherboard, it has been adaptable to each target we tested in a few hours of work. In addition, the use of spring-loaded pins has further removed the need for any soldering. These capabilities correspond to a *targeted attacker* defined by BitLocker's threat model (5.1).

Since the extracted secrets are chip-unique and immutable, an attacker can mount the glitching task, e.g., before the laptop reaches the end-user. Afterward, they only need (software- or hardware-based) SPI flash reading capabilities to mount the remaining attack.

6.1.2. Zen 1/+ (ROM attack). Due to an unpatchable vulnerability in the ROM boot loader of Zen 1 and Zen + CPUs [12], a much simpler setup, with no voltage glitching required, can be used to extract the keys (2.4.4). Instead, the attack only requires write capabilities for the BIOS flash chip. These can be achieved through either physical access with an SPI flash programmer or, in some cases, even through privileged *software-only* access. Although the proprietary NV storage format for Zen 1 and Zen + differs from the one we reverse-engineered, backporting 'amd-nv-tool' would merely be diligence work. The final attack would only involve opening the case and attaching the SPI flash programmer. Thus, we argue that this would lift the requirements of this attack to be suitable even for an *opportunistic attacker*.

6.2. Capabilities

There are two common FDE strategies employing a TPM an attacker could face. Depending on these strategies, our attack yields different capabilities for the attacker. Additionally, we discuss the capabilities an attacker gains against other TPM applications using our attack.

Our attack demonstrates that FDE using a *TPM-only* protector, i.e., a protector sealed with a PCR policy, can be decrypted regardless of the FDE implementation, which we verified by attacking Microsoft BitLocker (5.4).

The attack's capabilities against an FDE secured by a *TPM and PIN* protector rely heavily on the FDE implementation. In particular, the anti-hammering protections of the fTPM can be circumvented by our attack. However, as shown in our case study (5.3), a *TPM and PIN* protector can be implemented such that the security of a *PIN-only* strategy remains once the fTPM state is compromised.

This highlights the importance of BitLocker's *TPM and PIN* protector and emphasizes the importance of the used PIN strength, which we will elaborate in Section 6.4.1. We discuss general considerations regarding FDE implementations in Section 6.4.3.

In general, our attack gives an attacker access to the complete internal state of the fTPM. Since any authorization policy that specifies how and when a TPM object can be used or accessed is ineffective (4.4), it allows circumventing any security mechanism relying on the fTPM. For example, an SSH private key protected by a TPM (2.1.6) would be vulnerable to our fTPM state compromise.

6.3. Mitigations

Stopping arbitrary code execution attacks on the AMD-SP is the only way to mitigate our attack effectively. Unfortunately, this is currently not possible, as outlined in the following two Sections. Nevertheless, starting with Section 6.3.3, we propose soft measures that can make it harder to mount our attack.

6.3.1. Zen 2/3 (Glitch attack). Since our attack is based on Buhren et al.'s *voltage fault injection* attack [14], it is not easily mitigable for currently available CPUs. Mitigations involve changes in the hardware of the SoC or the ROM boot loader of the AMD-SP [14]. They can therefore be expected at the earliest in the next microarchitecture generation. This is an essential difference to attacks exploiting software vulnerabilities in the fTPM's code [19] or its execution environment [37]. Such attacks can be (and have been [2], [52]) mitigated with updates to the relevant software components.

A notable development in the realm of hardware-based FI countermeasures is Intel's introduction of digital detection circuitry into their Converged Security and Manageability Engine (CSME) [38], [39], which is Intel's counterpart to the AMD-SP. The CSME acts as the TEE for Intel's fTPM implementation, called Platform Trust Technology (PTT) [26]. With this approach, Intel preemptively follows Buhren et al.'s mitigation recommendations.

6.3.2. Zen 1/+ (ROM attack). The ROM boot loader vulnerability presented in [12] cannot be mitigated in existing Zen 1/+ CPUs, as it lies in the read-only memory, but has been mitigated since.

6.3.3. LSB secret extraction. On Zen 2 and below, we extracted the chip-unique secrets of the CCP (2.4.3) through unaligned AES operations (detailed in Section 4.3). These enabled us to derive the NV storage keys *offline*. While this extraction method cannot be mitigated on pre-Zen 3 CPUs, it has been mitigated in Zen 3 CPUs by disallowing the use of unaligned keys in AES operations. However, since we can still perform the first stage of the NV storage key derivation *online*, the measure could not mitigate our attack whatsoever.

6.3.4. Limiting hardware access. Our attack requires access to the SPI and SVI2 buses of the target system. Therefore, hardware obfuscation techniques could impede the attack, e.g., by hiding the relevant buses on the mainboard or making them otherwise physically inaccessible. However, we believe that these techniques will not effectively mitigate but only delay attacks. For example, if the SVI2 was inaccessible, an attacker could directly interfere with the mainboard's passive power supply components.

6.3.5. Software obfuscation. In the same vein, AMD could change the NV storage layout or its key derivation algorithm, e.g., by changing its constant value (Figure 6). Such obfuscation attempts can be circumvented by additional reverse engineering. In addition, on Zen 3, this would potentially also require extracting another secret (4.3). As there is no secret accessible to the AMD-SP's firmware that we cannot compromise through our attack, updated algorithms are not able to secretly generate, let alone use, keys to restore protection of the non-volatile data.

6.4. TPM Considerations

Both users and software and hardware developers can apply various strategies to alleviate the consequences of this attack or regain security for specific applications of a TPM. We will discuss these in the following.

6.4.1. Using BitLocker protectors effectively. Our attacks have shown that an fTPM cannot sufficiently protect its internal state against firmware or physical attacks. In such a scenario, a *passphrase-only* key protector of reasonable length provides better security than a *TPM-only* protector with a numeric PIN (5.3.1). This is in stark contrast to Microsoft's claim that "BitLocker provides the most protection when used with a Trusted Platform Module" [29] (see also in 2.3). In fact, of all available protectors (seen in Figure 1), *TPM-only* is arguably the weakest protection strategy.

However, one can make sense of Microsoft's claim differently: The TPM adds a layer of security when another factor is used. Specifically, a *TPM and PIN* protector is superior to a *passphrase-only* protection of the same length and character set. For example, assume an attacker has gained knowledge of the used BitLocker passphrase through social engineering and has physical access to the victim's machine. The attacker will then be able to boot up Windows and enter the correct PIN to get past the Windows pre-boot prompt. However, even though the BitLocker-protected volume is now unlocked, the attacker faces the regular Windows login prompt. At the same time, booting into an attacker-controlled system instead will not enable them to unseal the VMK: Even though they have obtained the correct PIN, changing the boot volume alters the PCR registers – a change the TPM will detect. In contrast, a *passphrase-only* protector would have enabled him to use, e.g., *Dislocker* to decrypt the protector and mount the volume.

6.4.2. Easing the setup of 'TPM + PIN' protectors. BitLocker's *TPM and PIN* protector is currently disabled by default. Users need to find the respective *Group Policy* settings (*Computer Configuration* → *Administrative Templates* → *Windows Components* → *BitLocker Drive Encryption* → *Operating System Drives* → *Require Additional Authentication at Startup*) before they can use these protectors. Additionally, they need to use the command-line tool 'manage-bde' to set them up [23]. Even worse, Microsoft BitLocker only allows so-called *enhanced PINs* (alphanumeric) with another *Group Policy* change (*Computer Configuration* → *Administrative Templates* → *Windows Components* → *BitLocker Drive Encryption* → *Operating System Drives* → *Allow enhanced PINs for startup*).

These advanced steps clearly show that Microsoft does not expect or encourage private users to use *TPM and PIN* protectors. For a non-technical user with high-security requirements, e.g., a journalist, BitLocker's default *TPM-only* configuration gives a false sense of security regarding their encrypted *data-at-rest*. While *TPM-only* protection relying on dTPMs is vulnerable to sniffing (3.1.4), our presented attack amends this capability for firmware TPMs.

We believe that it would significantly benefit the security of users if Microsoft provided a way to use *TPM and PIN* in their default BitLocker setup flow. It would also be thinkable to re-use or derive existing shared user secrets, e.g., the Microsoft Hello PIN, for this matter. However, *TPM and PIN* protection can be reduced to brute-forcing the PIN when a vulnerable fTPM is used (5.3). Since a non-technical user does not discriminate between discrete and firmware TPMs, it would be advisable for Microsoft BitLocker to suggest using an enhanced PIN when an fTPM is used, and implement NIST's password selection rules [17]. Furthermore, we deem it reasonable for Microsoft to distinguish the confusing nomenclature of an *enhanced PIN* from a numeric PIN by introducing a dedicated protector named *TPM and passphrase*, highlighting the improved security level.

6.4.3. Implementing FDE with *TPM and PIN* securely. Our case study shows that FDE implementations must employ standalone anti-brute-force measures beyond the sealed TPM object as BitLocker does (5.3.2). If the TPM is compromised, this upholds the protector's confidentiality to a degree a (non-TPM) PIN/password-only protector can achieve. The security of such a method dramatically depends on the length and complexity of the PIN or password, so strong requirements regarding its length and character set should be considered.

Effective authentication methods are often a trade-off between usability and (cryptographic) strength. In contrast to *password* and *passphrase*, the term Personal Identification Number (PIN) indicates weaker requirements regarding its length and character set. For example, credit card PINs often only consist of 4 numeric digits. The underlying smart card's lockout mechanism compensates for this low-entropy authentication factor that is potentially prone to brute-forcing attacks. The TPM's anti-hammering protection pursues a similar goal but is ineffective on fTPMs compromised by our proposed attack.

Theoretically, a layer of encryption could be added to the TPM specification: An object protected by a user authentication policy could be encrypted internally not only by a storage key derived from the parent object but also with a key derived from the user authentication string. However, this would break basic concepts of the TPM specification, e.g., it would no longer be possible to bind together a user authentication policy with another by an "or" clause. Additionally, the key derivation functions might also prove too expensive for dTPMs.

6.4.4. Firmware vs. Discrete TPM. Consider a TPM application relying exclusively on the TPM to seal a shared secret (like BitLocker's default configuration): Since our attack is arguably harder to mount than a dTPM bus sniffing attack, fTPMs can be considered more secure than a dTPM in this case. Apart from this particular case, dTPMs should be chosen over fTPMs for two reasons: Firstly, since dTPMs – to the best of our knowledge – have not been subject to full state compromises, they protect sensitive data that should never leave the TPM, e.g., private keys. Secondly, dTPMs uphold the protection of, e.g., the sealed BitLocker VMK protected by *TPM and PIN*, through the anti-hammering protection (see 5.3.1). This allows the literal use of low-entropy PINs (compare with Table 3). At the same time, dTPMs make it paramount to use second factors, as they otherwise disclose replayable secrets through their exposed communication interface (as seen with TPM-only BitLocker sniffing attacks). This significantly limits the practical usage of dTPM for server Full Disk Encryption (FDE), as every reboot, e.g., for scheduled security updates, would require human intervention.

7. Conclusion

AMD's firmware TPMs of recent microarchitecture generations are vulnerable to physical attacks such as voltage fault injection. They enable an attacker to access all assets secured by the TPM using low-cost, off-the-shelf hardware. To the best of our knowledge, our work is the first full TPM state compromise against a widely adopted TPM implementation. This is a considerable advance compared to previous attacks leveraging external communication to capture replayable secrets or sophisticated side channels compromising select parts of the TPM's internal state. Our full state compromise gives the powerful ability to defeat any TPM-based security: Applications relying exclusively on the TPM are left entirely unprotected, while those employing multiple layers of defense face the loss of their TPM-based security layer.

Motivated by Windows 11's push to use the TPM for even more applications, we apply the vulnerability to Microsoft BitLocker and show the first *fTPM*-based attack against the popular Full Disk Encryption solution. BitLocker's default *TPM-only* strategy manages – without any changes to the user experience – to swiftly step up a user's security in the face of a lost or stolen device. However, as our work complements the established attacks against dTPMs with an even more potent attack against AMD fTPMs, a *TPM-only* configuration lulls a non-technical user with high protection needs into a false sense of security.

When attacked with our full state compromise, BitLocker's *TPM and PIN* protector, in contrast, retains a security level according to the PIN's resistance against brute force. Nonetheless, we find fault that this feature is deeply buried inside Microsoft's *Group Policy* settings and hidden from a non-technical user. Moreover, a traditional PIN, i.e., a short numeric secret, does not provide even minuscule brute-force protection. Upgrading the security with so-called *enhanced PINs* – a euphemism for a *passphrase* – requires similarly advanced configuration changes.

Microsoft should empower their users to make an informed choice regarding the protection level of their *data-at-rest*: Users who fear a physical attacker with reasonable resources should opt for a *TPM and PIN* configuration. When BitLocker identifies that the underlying TPM is an fTPM, users should be urged to turn their *PIN* into a *passphrase*.

While TPMs are an essential tool to build secure applications, protect and manage cryptographic material, and anchor trust in the hardware of our physical devices, awareness of the required security levels and the TPM variant in use is essential. We hope that our contributions regarding the security of TPMs in general and AMD's fTPMs in particular guide users and developers on this journey.

Data Availability

We publish all code necessary to mount the attack under [27]. The repository further includes several intermediate results, e.g., flash memory dumps, to retrace the attack process without possessing the target boards and required hardware tools.

References

[1] Advanced Micro Devices. Ccp-dev.c - drivers/crypto/ccp/ccp-dev.c. Linux, 2017. https://elixir.bootlin.com/linux/v4.20.17/source/drivers/crypto/ccp/ccp-dev.c.

[2] Advanced Micro Devices. AMD Firmware TPM Updates. https://www.amd.com/en/support/kb/faq/pa-200, 2018.

[3] Advanced Micro Devices. AMD Memory Guard - Whitepaper. https://www.amd.com/system/files/documents/amd-memory-guard-white-paper-de.pdf, 2020.

[4] Advanced Micro Devices. Whitepaper: AMD Ryzen Pro 5000 Series Mobile Processor Security Features. https://www.amd.com/system/files/documents/amd-security-white-paper.pdf, 2021.

[5] E. Agostini and M. Bernaschi. BitCracker: BitLocker meets GPUs. *International Journal of Information Security*, May 2022. https://doi.org/10.1007/s10207-022-00589-4.

[6] Paul Alcorn. AMD Sets All-Time CPU Market Share Record as Intel Gains in Desktop and Notebook PCs. *Tom's Hardware*, February 2022. https://www.tomshardware.com/news/intel-amd-4q-2021-2022-market-share-desktop-notebook-server-x86.

[7] Denis Andzakovic. Extracting BitLocker keys from a TPM. https://pulsesecurity.co.nz/articles/TPM-sniffing.html, 2019.

[8] Aorimn. Dislocker. https://github.com/Aorimn/dislocker, April 2022.

[9] Will Arthur, David Challener, and Kenneth Goldman. *A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security*. A Practical Guide to TPM 2.0. Apress, Berkeley, CA, 2015. https://link.springer.com/content/pdf/10.1007/978-1-4302-6584-9.pdf.

[10] Benjamin Böck. Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker. https://img2.helpnetsecurity.com/dl/articles/windows7_firewire_physical_attacks.pdf, August 2009.

[11] Milan Brož. LUKS2 On-Disk Format Specification. https://gitlab.com/cryptsetup/LUKS2-docs/blob/main/luks2_doc_wip.pdf, July 2022.

[12] Robert Buhren and Alexander Eichner. All you ever wanted to know about the AMD Platform Security Processor and were afraid to emulate. https://www.blackhat.com/us-20/briefings/schedule/#all-you-ever-wanted-to-know-about-the-amd-platform-security-processor-and-were-afraid-to-emulate---inside-a-deeply-embedded-security-processor-20106, August 2020.

[13] Robert Buhren, Alexander Eichner, and Christian Werling. Uncover, Understand, Own - Regaining Control Over Your AMD CPU. https://media.ccc.de/v/36c3-10942-uncover_understand_own_-_regaining_control_over_your_amd_cpu, December 2019.

[14] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2875–2889, Virtual Event Republic of Korea, November 2021. ACM. https://dl.acm.org/doi/10.1145/3460120.3484779.

[15] Robert Buhren and Niklas Jacob. Glitching the AMD Secure Processor. PSPReverse, September 2021. https://github.com/PSPReverse/amd-sp-glitch.

[16] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1087–1099, London United Kingdom, November 2019. ACM. https://dl.acm.org/doi/10.1145/3319535.3354216.

[17] William E. Burr, Donna F. Dodson, Elaine M. Newton, Ray A. Perlner, W. Timothy Polk, Sarbari Gupta, and Ebad A. Nabbus. Electronic Authentication Guideline. Technical Report NIST SP 800-63-2, National Institute of Standards and Technology, November 2013. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf.

[18] Lily Chen. Recommendation for key derivation using pseudorandom functions (revised). Technical Report NIST SP 800-108, National Institute of Standards and Technology, Gaithersburg, MD, 2009. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf.

[19] Cfir Cohen. Full Disclosure: AMD-PSP: fTPM Remote Code Execution via crafted EK certificate. https://seclists.org/fulldisclosure/2018/Jan/12, 2018.

[20] Thomas Dewaele and Julien Oberson. TPM sniffing – Sec Team Blog. https://blog.scrt.ch/2021/11/15/tpm-sniffing/, 2021.

[21] Grigori Goronzy. Cryptenroll: Add support for TPM2 pin. https://github.com/systemd/systemd/commit/6c7a1681052c37ef354a000355c4c0d676113a1a, March 2022.

[22] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1229–1246, 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/han.

[23] Chris Hoffman. How to Enable a Pre-Boot BitLocker PIN on Windows. https://www.howtogeek.com/262720/how-to-enable-a-pre-boot-bitlocker-pin-on-windows/, 2017.

[24] Intel Corporation. Whitepaper: Strengthening Security with Intel Platform Trust Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/enterprise-security-platform-trust-technology-white-paper.pdf, 2014.

[25] Intel Corporation. Whitepaper: Intel® Total Memory Encryption. https://www.intel.com/content/www/us/en/architecture-and-technology/total-memory-encryption-security-paper.html, 2017.

[26] Intel Corporation. Code Sample: Protecting secret data and keys using Intel® Platform... https://www.intel.com/content/www/us/en/developer/articles/code-sample/protecting-secret-data-and-keys-using-intel-platform-trust-technology.html, March 2020.

[27] Hans Niklas Jacob and Christian Werling. PSPReverse/ftpm_attack on Github. https://github.com/PSPReverse/ftpm_attack, April 2023.

[28] Roger Lai. AMD Security and Server innovation. https://uefi.org/sites/default/files/resources/UEFI_PlugFest_AMD_Security_and_Server_innovation_AMD_March_2013.pdf, March 2013.

[29] Microsoft. BitLocker - Windows security. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview, December 2021.

[30] Microsoft. BitLocker Countermeasures (Windows 10) - Windows security. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-countermeasures, 2021.

[31] Microsoft. How Windows uses the TPM - Windows security. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm, 2021.

[32] Microsoft. Kernel DMA Protection (Windows) - Windows security. https://docs.microsoft.com/en-us/windows/security/information-protection/kernel-dma-protection-for-thunderbolt, 2021.

[33] Microsoft. Prepare your organization for BitLocker Planning and policies (Windows 10) - Windows security. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/prepare-your-organization-for-bitlocker-planning-and-policies, 2021.

[34] Microsoft. Trusted Platform Module (TPM) fundamentals (Windows) - Windows security. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/tpm-fundamentals, December 2021.

[35] Microsoft. Windows 11 Specs and System Requirements — Microsoft. https://www.microsoft.com/en-us/windows/windows-11-specifications, 2021.

[36] Microsoft. Manage-bde protectors. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/manage-bde-protectors, 2022.

[37] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2057–2073, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm.

[38] Daniel Nemiroff and Carlos Tokunaga. Tunable Replica Circuit for Fault- Injection Detection. https://i.blackhat.com/USA-22/Wednesday/US-22-Nemiroff-Fault-Injection-Detection-Circuits.pdf, August 2022.

[39] Daniel Nemiroff and Carlos Tokunaga. Whitepaper: Fault-Injection Countermeasures, Deployed at Scale. https://www.intel.com/content/www/us/en/architecture-and-technology/hardware-shield/fault-injection-countermeasures-white-paper.html, August 2022.

[40] OASIS PKCS 11 Technical Committee. PKCS #11 Cryptographic Token Interface Base Specification Version 2.40. http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html, April 2015.

[41] Lennart Poettering. Cryptenroll: Add support for TPM2 enrolling. https://github.com/systemd/systemd/commit/5e521624f292e2d469abe5e256db374dc17136ba, December 2020.

[42] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj.

[43] Bjorn Ruytenberg. Breaking Thunderbolt Protocol Security: Vulnerability Report. https://thunderspy.io/assets/reports/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf, April 2020.

[44] systemd. Systemd-cryptenroll manual page. https://www.freedesktop.org/software/systemd/man/systemd-cryptenroll.html#, 2022.

[45] systemd. Systemd-cryptenroll source code. https://github.com/systemd/systemd/tree/main/src/cryptenroll, May 2022.

[46] tpm2-software. Tpm2-pkcs11. https://github.com/tpm2-software/tpm2-pkcs11, April 2022.

[47] Trusted Computing Group. TPM 2.0 A Brief Introduction. https://trustedcomputinggroup.org/wp-content/uploads/2019_TCG_TPM2_BriefOverview_DR02web.pdf, June 2019.

[48] Trusted Computing Group. Trusted Platform Module Library Specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/, November 2019.

[49] Christian Werling, Robert Buhren, and Hans Niklas Jacob. PSPTool/psptool at master · PSPReverse/PSPTool. https://github.com/PSPReverse/PSPTool, 2022.

[50] Johannes Winter and Kurt Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 65(5):748–761, March 2013. https://linkinghub.elsevier.com/retrieve/pii/S0898122112004634.

[51] Philip Yip. BitLocker Installation Guide. https://dellwindowsreinstallationguide.com/bitlocker/, 2022.

[52] Zeljka Zorz. Intel releases updates to plug TPM-FAIL flaws, foil ZombieLoad v2 attacks. *Help Net Security*, November 2019. https://www.helpnetsecurity.com/2019/11/13/intel-fixes-dangerous-flaws/.